JBInsTrace: A Tracer of Java and JRE Classes at Basic-Block Granularity by Dynamically Instrumenting Bytecode

Pierre Caserta, Olivier Zendra

JBInsTrace: A Tracer of Java and JRE Classes at Basic-Block Granularity by Dynamically Instrumenting Bytecode

Pierre Caserta

LORIA - Campus Scientifique
615 Rue du Jardin Botanique CS 20101
54603 VILLERS-LES-NANCY CEDEX FRANCE
pierre.caserta@loria.fr

Olivier Zendra

INRIA Nancy - Grand Est / LORIA
615 Rue du Jardin Botanique, CS 20101
54603 VILLERS-LES-NANCY Cedex, FRANCE
olivier.zendra@inria.fr

Abstract

Understanding what happens during the runtime of a Java program is difficult. Tracking runtime flow can bring valuable information for program understanding and behavior analysis. Polymorphism, thread concurrency or even simple facts like the number of method invocations and the number of executed bytecodes are valuable information to track, but are difficult to compute outside the Java Virtual Machine (JVM) on running programs. In this paper, we present JBInsTrace, a new tool that instruments and traces Java bytecode. It produces static information about source code and a very fine grained trace of Java software execution, combining them to allow detailed analysis of the runtime. Our tool differs from others because it does not only trace program classes but also JRE classes, and does so at basic block level, without altering the JVM and without statically modifying class files. We explain JBInsTrace design, focused towards efficiency, which results in reasonable performance penalty.

Keywords: Java, Tracing, Dynamic, Bytecode, Instrumentation, Profiling, Program Analysis

1. Introduction

Dynamic analysis gives information on a particular execution of software and brings significant information to analyze and understand program behavior (Ball (1999); Cornelissen et al. (2009)), helping focus on modules which play a predominant role during the runtime of a program. Dynamic information such as reflection and dynamic class loading can be statically approximated (Christensen et al. (2003); Sreedhar et al. (2000)), but dynamic analysis is needed for a real accuracy (Hirzel et al. (2004)). On the other hand, static analysis gives insight about the source code itself and is thus useful to manage its

quality. Static analysis tools are more common than dynamic analysis tools; we think this is because static analyses are easier to implement.

One important goal of JBInsTrace[1] tool and this paper is thus to provide a technique to perform *fine-grained dynamic analysis* of Java software with reasonable performance penalty. To do this, we divide profiling in two stages. The first stage consists only in tracing the exact runtime control flow and saving static information about the source code. The second stage performs an off-line analysis of the trace (a.k.a. post mortem trace analysis) and of the static information from the previous step, in order to create the call graph and to compute complex dynamic metrics (Dufour et al. (2003); Arisholm et al. (2004)). One advantage of this two-stage technique is that metrics are not computed at runtime. In addition, when new metrics have to be computed, the instrumentation does not need to be modified.

Most of the work on dynamic analysis focuses on tracing runtime at the method level. Even though some works are very consistent in tracking method flow and do provide a lot of valuable information, we wanted the extra accuracy of a finer level. Our analysis of Java programs is thus made at the *basic block*[2] level, which means that execution is tracked even within methods. We can also analyze call sites within basic blocks.

Our tracing tool JBInsTrace instruments program classes as well as JRE classes because we want a full coverage of the whole runtime of the program, to obtain the most complete trace of the control flow, which we think is necessary to fully understand the behavior of a program. Of course, JBInsTrace allows switching off the tracing of JRE classes when desired.

The emergence of new paradigms such as Aspect-oriented programming (AOP) makes it easier to do rapid prototyping of profilers, debuggers, tracers, and reverse engineering tools (Villazon et al. (2009)). Indeed, dynamically adding new functionalities on running programs by modifying classes at runtime through bytecode instrumentation is a convenient way to add behavior to programs. Our tracing tool and technique rely on this paradigm to instrument Java programs at strategic join points and extract information from their runtime. However, the main difficulty in instrumentation lies in finding an efficient design for the profiler (Mock (2003)).

In this paper, we present the design and some implementation details of our dynamic instrumentation and tracing technique. In section 2 we give an overview of the technique and tool we created to dynamically instrument Java classes. Section 3 details our technique. Then, in section 4 we discuss the static and dynamic outputs of the tracer and how they are analyzed to compute valuable information. Section 5 shows performance results that validate our approach to take user-perceived efficiency into account from the very beginning. Finally, section 6 concludes and presents future work.

2. Overview of the JBInsTrace tracing technique

Since version 1.5, Java provides services that allow Java agents to instrument programs running on the JVM. The agent is executed in the same JVM, loaded by the same class loader, and governed by the same security policy and context as the program. We

[1] Available at: http://www.loria.fr/~casertap/jbinstrace.html

[2] A basic block is a sequence of bytecode instructions which ends by a jump instruction.

thus intercept Java classes when they are loaded to instrument them "on-the-fly", even if the program has custom class loaders.

The design of the JRE is complex, with a lot of coupling between classes. Mechanisms to realize simple operations thus imply many underlying method calls and execute many bytecodes. This make understanding what happens when calling a JRE method difficult. As an example, the first program most people write is "hello_world", coded in Java as: `System.out.println("Hello world ...");`. The runtime of this Java program executes around 42550 bytecodes, a major part of which are program initialization bytecodes. The `println` method call alone generates about 160 method calls and executes around 2660 bytecodes. I/O methods are known to be complex, but this example shows the importance of JRE methods execution.

A specificity of our profiling technique is that it performs a fully dynamic instrumentation on all Java classes, including core JRE classes (see section 3.1). Classes loaded because of reflection are also treated like the ones automatically loaded by the JVM: code `Class.forName("X")` causes class `X` to be loaded and initialized, hence instrumented.

Since JRE classes may be used by both the observed program and the tracer, instrumented JRE classes could lead to trace pollution by the tracer. We explain our mechanism to avoid this pollution in detail in section 3.3. However, like any Java instrumentation technique injecting bytecodes into classes, ours still impacts timing and thread scheduling because of the extra tracing bytecodes to execute, even though they are not put in the trace.

Runtime flow is traced at basic block level. This granularity gives insight about intraprocedural control flow. With our JBInsTrace trace, we know how many times every single bytecode is executed. Since call sites are traced, information about polymorphism can be extracted during the analysis step by comparing the call site static type and the method called at runtime. Native method calls and thread switches are detected as well and feed additional information to our tracer (see section 3). In fact, JBInsTrace provides enough information to precisely re-build the whole runtime flow of the profiled program and thus perform very precise dynamic analyses.

Figure 1 summarizes the global functioning of our profiling technique. When a class is loaded by the `ClassLoader`, a Java agent catches the bytecode of this class on the fly (see section 3). The `Instrumentor` parses the class, injects new bytecodes and saves static information about basic blocks, methods and classes (see section 3.2). The injected bytecode adds a new (tracing) behavior, without altering the functional semantics of the original bytecode but with an impact on timing and scheduling (see section 3.2). At runtime, the injected bytecode calls back our `Tracer` (see 3.2.3), which must take care not to pollute the trace with tracer events (see section 3.3). At the end of the runtime, all the static information and the dynamic execution trace are available in files, ready to be processed by any appropriate analyzer (see section 4).

3. Dynamic instrumentation of classes

Our dynamic instrumentation transforms only classes which are actually used during a specific execution, without altering the original `.class` bytecode files on disk. We made this choice because static instrumentation would have implied transforming all classes that *might* be used by the program, whereas only a small percentage of this huge set

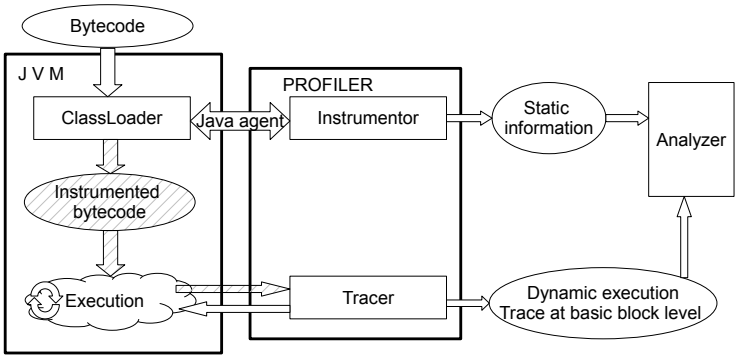

Figure 1: Functioning of our profiler

of classes would actually be used at runtime. Moreover, static instrumentation requires keeping both an instrumented and a non-instrumented version of class files on disk, more than doubling the space. Since classes appear in JAR files, it would have been constraining to first extract classes, then instrument them and finally re-JAR them, and do so at each new version of a particular class. Furthermore, static instrumentation cannot be done on dynamically created classes. For all these reasons we decided to instrument code dynamically.

The following three subsections detail our technique. Section 3.1 explains how the Java agent works. Section 3.2 details how we instrument classes. Section 3.3 shows how we avoid polluting the trace of the running program with our JBInsTrace tracer.

3.1. Instrumenting classes with a Java agent

A Java agent, deployed as a JAR file, is a service that allows instrumenting programs running on the JVM through the `-javaagent` command line option. Our JBInsTrace tracer has been tested with the Java HotSpot VM and the IBM z/VM, but it should work on any JVM that provides the agent service.

The `premain` method is a method called by the Java agent before the `main` method of the application but after the JVM has initialized. JBInsTrace uses `premain` to register its class transformer (`Instrumentor`) which will be applied on all future class definitions.

However some classes are loaded during the initialization phase of the JVM (we name them "core JRE classes"), hence before our class transformer is registered. Indeed, the JVM follows a lazy loading process, which means that classes are loaded only when needed. When the JVM starts without the Java agent, it loads the initial class, initializes it, and invokes its class method `main(String[])` (Lindholm and Yellin (1999)). With the Java agent, the initialization process is changed and all the classes needed for the execution of the `premain` method are also loaded by the JVM during its initialization. In our case this adds around 350 core JRE classes, mainly from the `java.lang` package.

We instrumented such core JRE classes with the `retransform` method from Sun-Oracle implementation, but had to overcome some limitations: the instrumentation must not add, remove or rename fields or methods, change the signatures of methods, or change inheritance (Sun Microsystems (2008)).

This differs from other existing dynamic techniques which usually introduce new methods acting as wrappers and adding extra arguments to methods to track the calling context (Villazon et al. (2009)). With such techniques, core JRE classes have to be treated differently. The solution proposed by Binder and al. (Binder et al. (2007)) in their "JP" tool was to statically instrument the `.class` files. They have reimplemented their tool, now called "JP2" (Sarimbekov et al. (2011)), which is much more advanced and does not suffer from many of the restrictions the older version imposed. The new tool does not perform structural change on classes anymore but still has to statically modify the `Thread` class. Instead of adding new parameters to methods, they add a new field to the `Thread` class and use it as a reference to save the calling context tree of the running program. To trace native calls, they still need to statically instrument all the JRE classes.

Obviously, with bytecode instrumentation, no tracing occurs during the JVM initialization phase. The only way to do such tracing would be by modifying the JVM.

3.2. Bytecode instrumentation

Bytecode instrumentation injects new bytecodes into class. Several tools and libraries exist to do this. We chose the ASM framework (Bruneton et al. (2002a,b); Bruneton (2007); Kuleshov (2007a)) which provides libraries to parse Java classes and add bytecode, allowing complex transformations, with very low memory requirements. According to (Kuleshov (2007b)), it is much faster than other bytecode transformers such as BCEL (Dahm et al. (2002, 2001)), SERP (White (2002)) and Javassist (Chiba (2004)).

In subsection 3.2.1 we show how we extract static information from loaded classes. Then in subsection 3.2.2 we describe where we inject bytecodes in classes and what the purpose of these new bytecodes is.

3.2.1. Static information extraction

When classes are loaded, JBInsTrace parses them and performs three operations:

- unique identifiers are assigned to each class, method and basic block.
- static information is extracted from the class source code.
- the class is instrumented with new bytecodes.

These identifiers are used to identify parts of the source code, when the trace is saved on disk. All the important static information about classes, methods and basic blocks is also saved. This information is then used during the analysis phase, when identifiers are read from the trace and related the corresponding static information.

The static information is saved in XML (with the Xerces library) or CSV format. In this paper we use XML for the static information because it is more readable. Here is an example of the static information saved for a basic block:

```
<basic_block id="5095440"
           metrics="4 0 0 0 1 1 0 1 0 0 0 0 0 0 0 1 0 1 0">
       <call_site_list>
              <call_site call="java/nio/CharBuffer:hasRemaining()Z"
                         type="INVOKEVIRTUAL"/>
       </call_site_list>
</basic_block>
```

The `metrics` field is a list of basic block metrics. The latter contains the numbers of: bytecodes, array read operations, array write operations, floating point operations, reference loads, control bytecode instructions, changing control bytecode instructions, read operations on object fields defined in the class, write operations on object fields defined in the class, read operations on primitive field defined in the class, write primitives on object field defined in the class, read operations on object field defined in another class, write operations on object fields defined in another class, read operations on primitive fields defined in another class, write primitives on object fields defined in another class, invocations of another class, new bytecode operations, virtual calls and finally static calls. In future versions of JBInsTrace we plan to save the entire bytecode of each basic block instead of just a list of metrics. More information can be found on the JBInsTrace web page (http://www.loria.fr/~casertap/jbinstrace.html).

3.2.2. Code segmentation

JBInsTrace traces 3 kinds of events: when a method starts and ends, like any other tracing tool, but also when a basic block is executed. Because a basic block contains only one jump instruction at its end, it is executed from beginning to end, unless an exception occurs. Section 4.2 explains how JBInsTrace follows the runtime flow with exceptions.

An event is recorded as a single integer to maximize performance. Figure 2 shows how we divide a 32-bit integer, named *event number*, to encode 3 pieces of information:

- The 2 most significant bits encode the *event type identifier*: method start, method end, basic block execution.
- The 19 bits in the middle encode the unique identifier of the method.
- If the event type is the execution of a basic block, the 11 least significant bits represent the unique identifier of the basic block within the method (unused otherwise).

Note that the *event number* can be negative, since the most significant bit is used for *event type* encoding.

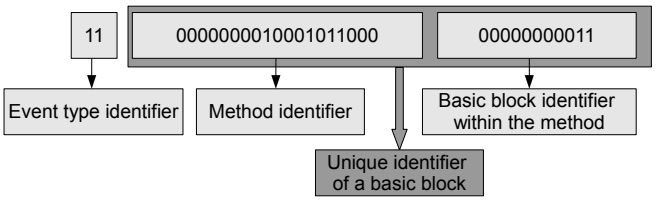

Figure 2: Event number structure

Together the *method identifier* and the *basic block identifier* form a unique identifier for the basic block. In fact, every *event number* is unique. This coding can identify 524288 different methods that can have at most 2048 basic blocks each. We think these limitations are reasonable. Indeed, ArgoUML, which is a large open source UML modeling tool loading around 5100 classes at startup, counts around 12000 instrumented methods, with at most 180 basic blocks in a method. Moreover, trying about 100 runs of ArgoUML with many different inputs led to a cumulated number of used methods of

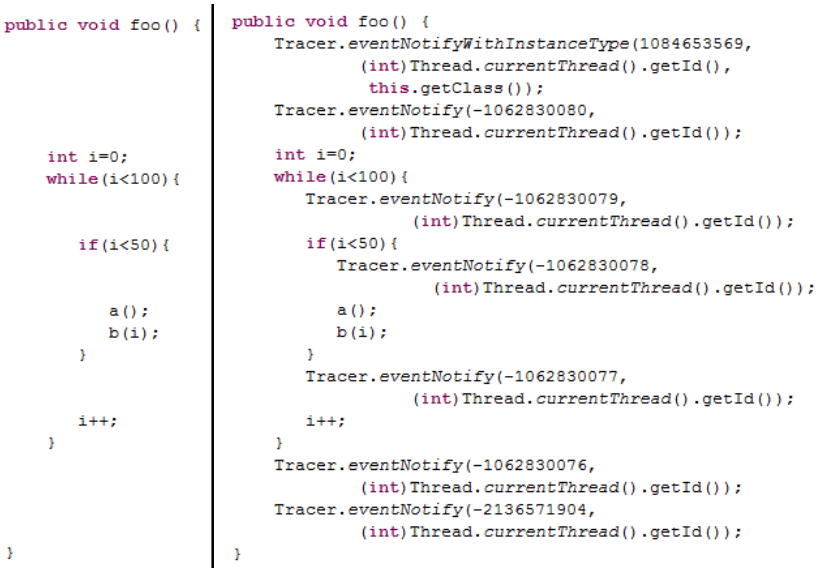

Figure 3: Java bytecode instrumentation principle (pseudo-Java code)

50000. If needed, we could switch to using a 64 bit long integer to lift these limitations, but this would increase memory usage.

3.2.3. Instrumented code

Java programs are multi-threaded so runtime flow has to be put in the context of the actual thread where execution takes place. Our tool thus also records for each event (method start, method end, basic block start) the running thread number. To do it, every time an event occurs, the instrumented code calls the `Tracer` (via the `notifyEvent` static method) and passes it the associated *event number* and the current *thread id*. The Java code corresponding to our instrumented bytecode is:

```
Tracer.eventNotifier(eventNumber, (int)Thread.currentThread().getId());
```

Figure 3 shows an example of how a piece of source code is instrumented. The left part 3(a) is the Java code before instrumentation, while the right part 3(b) represents the source code after instrumentation.

As illustrated in figure 3, instrumentation is done:

- at the beginning of each method. There, `eventNumber` is composed of event type '01' binary meaning 'method start', followed by the method identifier.
- at the end of each method. There, `eventNumber` is composed of event type '10' binary meaning 'method end', followed by the method identifier.

- at the beginning of each basic block. There, `eventNumber` is composed of event type '11' binary meaning 'basic block start', followed by the method identifier, followed by the basic block identifier within the method.

With this technique, the numerous calls to the `Tracer` cost time, but we observed good performance results (see section 5). Since it adds no new method nor any extra parameter, it preserves the semantics of the original program and the injected code is the same for all classes, including JRE core classes. However, like any technique which injects new bytecodes into classes, ours impacts timing and thread scheduling.

3.2.4. Saving dynamic instance types and class loading information

Class identifiers have two purposes: depict the dynamic type of an instance on virtual and interface call sites, and identify the loaded classes.

To depict the dynamic type of instances on virtual and interface calls, the instrumented code is slightly different. When a method starts, the receiver dynamic type is obtained using the reflexive method `getClass():java.lang.Class` on `this` (Sundaresan et al. (2000)). Then the instrumented code calls method `eventNotifierWithInstanceType` of the `Tracer`, which saves the corresponding class identifier in the output trace.

The pseudo-Java code inserted at the beginning of methods is thus:

```
Tracer.eventNotifierWithInstanceType(eventNumber,
                (int)Thread.currentThread().getId(), this.getClass());
```

When a class is loaded, the `Instrumentor` notifies the `Tracer` which saves this information in the trace buffer. The event number is composed of event type '00' binary, followed by the identifier of the loaded class.

3.3. Trace neutral tracer

Because both program and tracer run on the same JVM, JBInsTrace uses many JRE classes which are themselves instrumented. We must separate the execution of the tracer and the execution of the program to avoid having traces of the tracer in the trace files. To solve this issue, we resorted to a system of boolean flags that switch tracing on or off. These flags avoid creating infinite recursion between JRE classes and the `Tracer`.

First, instrumented bytecode calls the `Tracer` using the `notifyEvent` method. The `Tracer` changes the `authorization` boolean to `false` to forbid tracing. Then the code of our `Tracer` is executed. This code relies on the JRE classes, which are themselves instrumented, causing these JRE classes to call the `Tracer` again. This time the `authorization` flag is `false` and forbids tracing, since we are in the `Tracer`. At the end of the `notifyEvent` method, the `authorization` flag is switched back to `true` to allow tracing again.

The same boolean mechanism is used by the `Instrumentor` to forbid its own tracing, as shown in figure 4.

The principle is trivial but care must be taken because Java programs are multi-threaded, so we must constantly track the number of the current thread and manage one `authorization` flag per thread.

Figure 5 presents the UML class diagram of our JBInsTrace tracer.

The `LockManager` class plays a central role in the design of the tracer. It provides a way to differentiate, in the executed bytecodes, those from the program itself and those

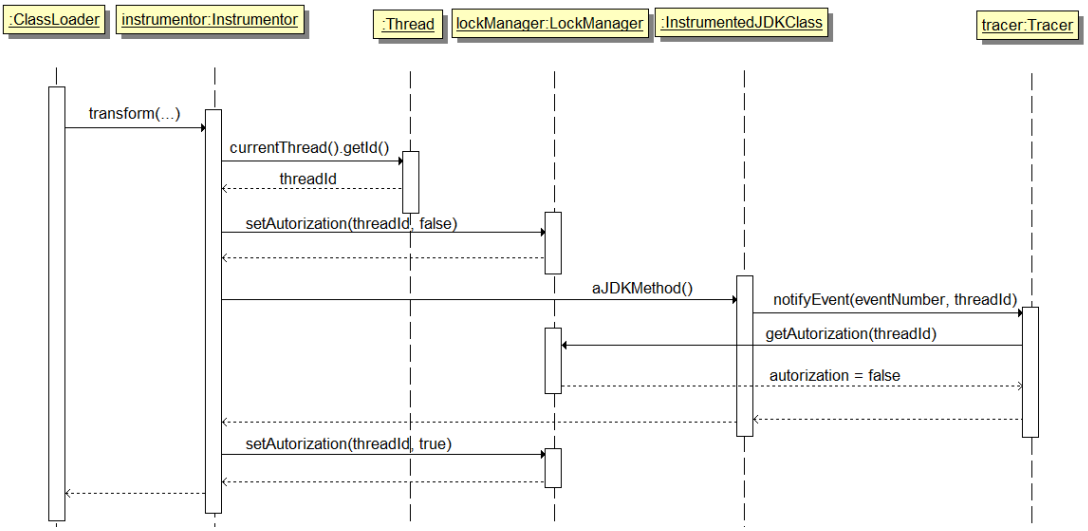

Figure 4: Sequence diagram illustrating the technique used to prevent trace pollution

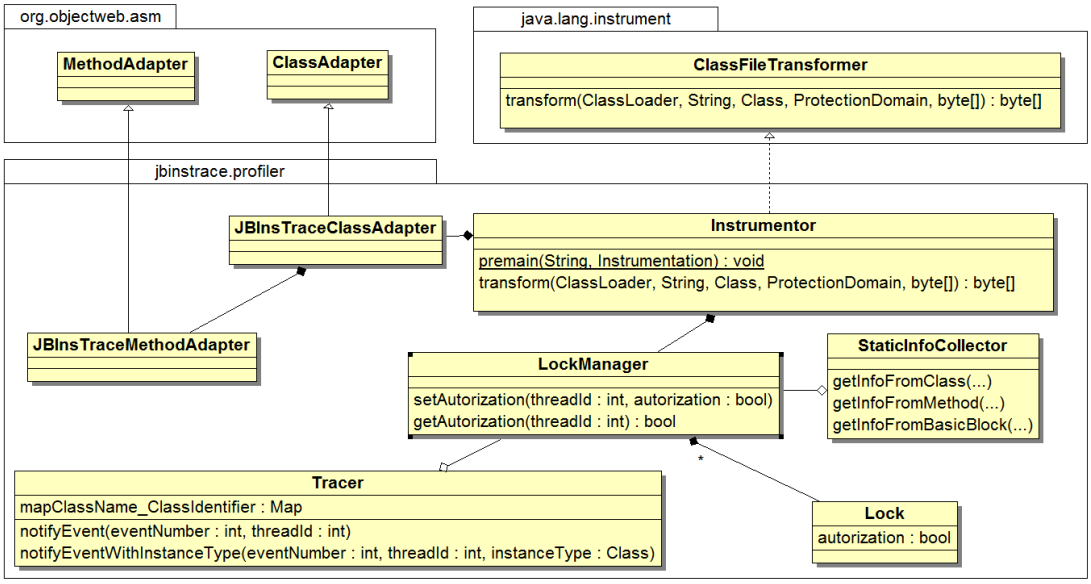

Figure 5: UML Class diagram of our JBInsTrace tracer.

from the tracer. This class stores one `Lock` object for each thread of the program and each `Lock` object contains the `authorization` boolean. The `getAuthorization` method returns the appropriate boolean corresponding to the current thread.

3.4. The Tracer

Our `Tracer` class is a singleton which can be called by any class of the program. This `Tracer` simply buffers the list of observed events (integers) before writing them to disk. The `Tracer` can manage the event buffer using two modes:

Mode 1 − one unique, shared trace to save the pairs (eventNumber, threadId) for all threads.

Mode 2 − one trace of events per thread (no sharing),

In mode 1 (one unique trace for all threads), synchronization is mandatory to prevent pollution and/or dead locks. However, this may drastically slow down the tracer because of waits to reach the critical zone. The tracer in mode 1 (one unique trace) is thus much slower than in mode 2 (one trace by thread), but it preserves the thread concurrency scheduling.

Note that the `Tracer` is called intensively by the instrumented program so methods `notifyEvent` and `notifyEventWithInstanceType` must be as fast as possible. In fact the only operations in these methods are:

- check the value of the `authorization` boolean that corresponds to the current thread (see figure 4)
- save the `(event number, thread ID)` pair in the shared buffer (in mode 1), or save only the `event number` in the buffer according to `the thread ID` (in mode 2)
- in `notifyEventWithInstanceType` an extra operation consists in saving the dynamic instance type in the appropriate buffer.

4. Exploiting the program trace and static information

Our JBInsTrace tracer provides files that contain the exact control flow of each thread at basic block level, plus static information about these basic blocks. We exploit this information with a trace analyzer (see the rightmost part of figure 1). In this section we explain how to develop such a trace analyzer for JBInsTrace traces. In subsection 4.1 we explain how to build a new JBInsTrace trace analyzer. Then in subsection 4.2 we describe how the tracing of exceptions is handled in JBInsTrace. Finally, subsection 4.3 details how we detect native calls.

4.1. Building a JBInsTrace Trace Analyzer

The first step is to read the static information and construct the data structure representing the observed program. An interesting meta-model of Java programs can be found in (Hoffmann et al. (2008)): since it takes basic blocks into account it is adapted to model the static information from JBInsTrace. We used a similar meta-model in our analyzer. Then, a map that links the integer identifiers of the JBInsTrace trace (see section 3.2.2) to their corresponding elements in the meta-model instance has to be built.

The functioning of our analyzer is as follows. We relate each event of the trace with the static information saved. For instance if we find that basic block b has been executed

in the trace, we search for the static information of basic block b and see what kind of bytecode basic block b has. This technique allows to re-simulate the whole execution call stack and to compute very precise dynamic metrics. For example, for each method call event in the trace, we can compare the static type of a virtual call site (in the static information) and the dynamic type of the instance (in the trace), to compute metrics related to polymorphism such as the metric that counts how many times receiver types change on call sites.

The authors of (Dufour et al. (2003)) have defined several interesting dynamic metrics which are implemented in our analyzer. These metrics can be used to categorize programs according to their dynamic behaviour in five areas: size, data structure, memory use, concurrency, and polymorphism. Dynamic metrics related to program size and structure, use of data structures, use of polymorphism, memory footprint and concurrency are computable with our analyzer. We also use it to compute precise call graphs of programs and libraries, to feed in our advanced visualization tool (Caserta et al. (2011)).

4.2. Tracking exceptions

The only way the execution flow of a basic block may be disrupted is by an exception, which causes program control to jump to an exception handler. In some cases, the call stack has to be popped until the corresponding `catch` block is found. The `try/catch` structure thus has to be recorded, to have each basic block aware of the basic block identifiers it can potentially jump to. This information allows simulating the call stack during the analysis phase.

Here is an example of the static information saved on a basic block with two possible exception handlers:

```
<basic_block id="5095441"
            metrics="7 0 0 1 0 1 0 1 0 0 0 0 2 0 0 1 0 1 0">
 <exception_handler_list>
  <exception_handler basic_block_id="3764234"
                         type="java/io/InterruptedIOException" />
       <exception_handler basic_block_id="3764235"
                         type="java/io/IOException" />
 </exception_handler_list>
</basic_block>
```

4.3. Tracking native calls

With our tracing technique, it is the called method that informs the trace about its beginning and its end, not the caller. Native method calls are traced differently, because native methods are not coded in Java, hence are not instrumented. Moreover our technique does not add any new method to classes, because of the limitation on core JRE classes. Using wrapper methods to detect native calls is thus impossible for us. As a matter of fact, we use a very simple technique to detect native calls. When analyzing runtime, we still follow static information about each basic block. When a method call is found in the basic block static information, we check whether the corresponding call is present in the actual dynamic trace. If not, we know this call is a native one. Exceptions thrown in native methods are tracked like other exceptions (see section 4.2).

5. Performance

In this section, we show the time performance of our JBInsTrace v0.6 tracer on several Java benchmarks. We first present results obtained on well-known Open Source Java software, then results on the SPECjvm2008 benchmark suite. Our tests were made on a 8-core, 64-bit Intel(R) Xeon(R) CPU E5440 2.83GHz workstation, with 16GB RAM, running Windows Vista SP1. All tests were performed with our tracer in mode 2 (one trace by thread).

The benchmarks are:

- ArgoUML v0.28, the leading Open Source UML modeling tool including support for all standard UML 1.4 diagrams. 5173 classes are loaded during startup.
- JEdit v4.3pre17, a mature programmer text editor with hundreds of person-years of development behind it, including developing plug-ins development. 2822 classes are loaded during startup.
- Columba v1.0, an email client written in Java, featuring a user-friendly GUI with wizards and internationalization support. The runtime of the startup of this interactive benchmark loads 3646 classes.
- Apache Ant v1.8.2, a Java library and command-line tool to drive processes described in build files as targets and extension points dependent upon each other. Ant is mainly used to build Java applications. The runtime of the startup of Ant (performing a compilation of itself) loads 1438 classes. We chose to build Ant itself because it does activate the most common features that are used to build a typical java project according to (Andy Zaidman (2008)).
- SPECjvm2008, a benchmark suite that focuses on the performance of the JRE executing a single application; it reflects the performance of the hardware processor and memory subsystem (Shiv et al. (2009)). The SPECjvm2008 workload mimics a variety of common general purpose application computations (Benchmarks (2008)).

An issue with instrumentation is the instrumented program slowdown. In JBInsTrace, three elements contribute to this: the instrumentation process itself, the instrumented bytecode which calls the tracer back, and the recording of the static information and trace to the disk.

Our JBInsTrace tool has two ways of writing the trace files to disk. When memory is constrained, the trace is incrementally written to the disk, at "stop-the-world" garbage collector times. This allows a larger trace to be built, without having to retain it all in memory. Without memory constraints, the entire trace can be kept in memory and dumped to disk at the very end of the execution.

Table 1 presents performance results on ArgoUML, JEdit, Columba and Ant, showing the impact of JBInsTrace on the benchmark runtime. Our goal is to demonstrate that the instrumented software is still fully usable even with our tracer switched on. The first three benchmarks are interactive programs, but to avoid disturbing performance measurements with human interaction and for the sake of repeatability we only traced their startup. The second column in the table shows the benchmark execution time without tracer. The 3rd and 4th columns show the runtime figures with the tracer on.

Column 3 is the execution time, while column 4 is the time to save data to disk (we had enough memory to keep the entire trace in memory and dump it at the end of the runtime).

Software	Without the tracer	With the tracer	
		Execution	Recording
ArgoUML	9	44	152
JEdit	6	14	15
Columba	7	17	19
Ant	5	40	170

Table 1: Impact of our JBInsTrace tracer (execution times, in seconds).

Regarding total execution times with and without the tracer, the slowdown factor is 21.8 for ArgoUML, 4.8 for JEdit, 5.1 for Columba and 42 for Ant. Obviously, there is still work to do to improve the performance of our tracer.

But speed is not our main goal: we first want to acquire a very detailed trace of the execution, including JRE classes, while keeping the tool easy to use and very flexible regarding the information that can be collected at runtime. This of course implies some performance penalty.

However, looking at how the different stages contribute to the slowdown in Table 1 shows that most of the time is spent saving data to disk: the larger the trace saved to disk, the higher the slowdown factor. This explains the difference of performances between these benchmarks, because slowdown factors correspond to the trace sizes magnitude. Note that the size of the trace depends on the size of the scenario and operation performed in the software. The time to write the data to disk depends strongly on hardware. For example, it should be significantly lower with a SSD hard drive.

Although the time spent to write to disk may not be neglected, is it important to remember that our main goal is to maintain a good perceived user experience, keeping the instrumented software fully usable at runtime, even when the tracer runs.

Moreover, in our tests the trace fitted in memory until the end of the tracing, so recording did not bother the user during the execution of the software per se.

For larger software systems, nonetheless, the trace does not fit in memory. In such cases, JBInsTrace has to periodically pause the execution to dump the trace to disk.

When the time spend to write data to disk is factored out, the — perceived — slowdown factors during the execution phase are 4.9 for ArgoUML, 2.3 for JEdit, 2.4 for Columba and 8 for Ant. This shows that the software remains usable with our tracer switched on, which is further confirmed by usability tests performed with actual users.

Figure 6 presents JBInsTrace performance on the famous SPECjvm2008 benchmark suite. These are much smaller, non-interactive and computationally intensive benchmarks. The overhead is computed as $\frac{operations}{second}$ and the slowdown factor is computed as $\frac{operations/second\ without\ profiling}{operations/second\ with\ profiling}$. For this experiment the time to write the trace to the disk at the end of the profiling is not accounted. The tests were performed with both HotSpot Client and HotSpot Server JVMs, with little difference in the results.

The overall results with SPECjvm2008 are consistent with the ones obtained with our 5 larger Java applications, although the intensity of computations degrades performance

a bit. The sunflow benchmark is a pathological case, not very representative of the actual performance of JBInsTrace, that we have to investigate.

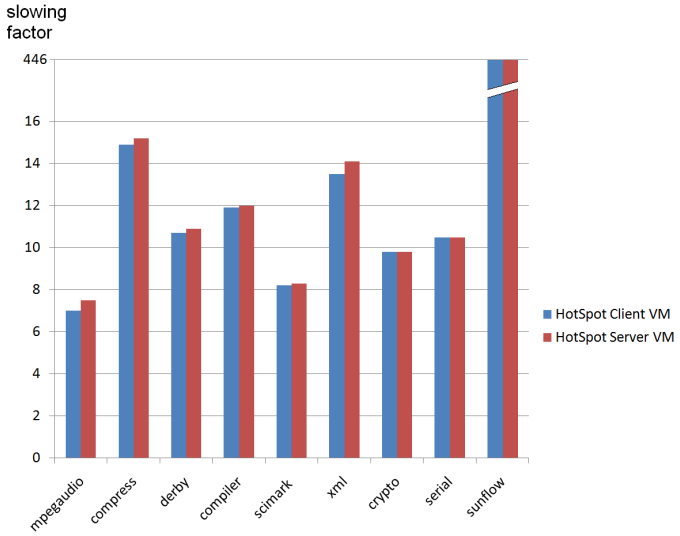

Figure 6: Performance results on the SpecJVM Benchmark.

6. Conclusion and future work

In this paper, we presented our tracing technique and tool, JBInsTrace. JBInsTrace makes it possible to trace what happens at runtime in Java programs at basic block level, on program classes as well as JRE classes using only dynamic bytecode instrumentation.

This complete tracing including JRE classes, at basic block level, generates a very detailed and complete trace of the runtime, representing a huge amount of information to process. Experimental results show that our tracer has a reasonable performance penalty. However pure speed is not our main goal: perceived user experience is more relevant and JBInsTrace provides a good one. Furthermore, JBInsTrace is mainly a research-oriented tool; nonetheless its scalability limits can be found in industrial software (Zaidman (2006)).

Our implementation is based on the Java agent service bundled with Java 1.6. As a result, JBInsTrace is very easy to install and ready to use, with no impact on existing Java library files, which makes life easier for users.

Several ways exist to improve our technique and tool. The first direction of our future work will imply performing extensive benchmarking of Java programs with our tracer to gather broader results. The second direction will add new methods in classes when limitations to transform core JRE classes will be lifted. We will especially add a `finalize()` to every class of the program, to detect and trace object destructions when they occur. Another future direction concerns how to effectively exploit runtime information to have a better understanding of software runtime and to optimize programs.

7. References

Andy Zaidman, S. D., 2008. Automatic identification of key classes in a software system using webmining techniques. Journal of Software Maintenance and Evolution: Research and Practice 20 (6), 387–417.

Arisholm, E., Briand, L., Foyen, A., 2004. Dynamic coupling measurement for object-oriented software. Software Engineering, IEEE Transactions on 30 (8), 491–506.

Ball, T., 1999. The concept of dynamic analysis. In: Software EngineeringESEC/FSE99. Springer.

Benchmarks, S., 2008. Standard performance evaluation corporation. http://www.spec.org/jvm2008/.

Binder, W., Hulaas, J., Moret, P., 2007. Advanced Java bytecode instrumentation. In: Proceedings of the 5th international symposium on Principles and practice of programming in Java. ACM, pp. 135–144.

Bruneton, E., 2007. Asm 3.0 a java bytecode engineering library. URL: http://download. forge. objectweb. org/asm/asmguide. pdf.

Bruneton, E., Lenglet, R., Coupaye, T., 2002a. ASM: a code manipulation tool to implement adaptable systems. Adaptable and extensible component systems.

Bruneton, E., Lenglet, R., Coupaye, T., 2002b. Asm: a code manipulation tool to implement adaptable systems. Adaptable and extensible component systems 30.

Caserta, P., Zendra, O., Bodénès, D., Sep. 2011. 3D Hierarchical Edge Bundles to Visualize Relations in a Software City Metaphor. In: 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2011). Williamsburg, États-Unis, pp. 1–8.

Chiba, S., 2004. Javassist: Java bytecode engineering made simple. Java Developers Journal 9 (1).

Christensen, A., Møller, A., Schwartzbach, M., 2003. Precise analysis of string expressions. Static Analysis, 1076–1076.

Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., Koschke, R., 2009. A systematic survey of program comprehension through dynamic analysis. Transactions on Software Engineering 35 (5), 684–702.

Dahm, M., van Zyl, J., Haase, E., 2002. The bytecode engineering library (BCEL).

Dahm, M., et al., 2001. Byte code engineering with the bcel api. Java Informationstage 99, 267–277.

Dufour, B., Driesen, K., Hendren, L., Verbrugge, C., 2003. Dynamic metrics for Java. ACM SIGPLAN Notices 38 (11), 149–168.

Hirzel, M., Diwan, A., Hind, M., 2004. Pointer analysis in the presence of dynamic class loading. ECOOP 2004–Object-Oriented Programming, 96–122.

Hoffmann, B., Pérez, J., Mens, T., 2008. A case study for program refactoring.

Kuleshov, E., 2007a. Using ASM framework to implement common bytecode transformation patterns. Proc. of the 6th AOSD, ACM Press.

Kuleshov, E., 2007b. Using the asm framework to implement common java bytecode transformation patterns.

Lindholm, T., Yellin, F., 1999. Java virtual machine specification. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Mock, M., 2003. Dynamic analysis from the bottom up. In: WODA 2003 ICSE Workshop on Dynamic Analysis. Citeseer, p. 13.

Sarimbekov, A., Moret, P., Binder, W., Sewe, A., Mezini, M., 2011. Complete and Platform-Independent Calling Context Profiling for the Java Virtual Machine. In: Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation. pp. 1–15.

Shiv, K., Chow, K., Wang, Y., Petrochenko, D., 2009. SPECjvm2008 performance characterization. Computer Performance Evaluation and Benchmarking, 17–35.

Sreedhar, V., Burke, M., Choi, J., 2000. A framework for interprocedural optimization in the presence of dynamic class loading. In: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation. ACM, pp. 196–207.

Sun Microsystems, I., 2008. Java platform standard ed. 6. package java.lang.instrument.

Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., Godin, C., 2000. Practical virtual method call resolution for java. In: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. ACM, pp. 264–280.

Villazon, A., Binder, W., Moret, P., 2009. Flexible Calling Context Reification for Aspect-Oriented Programming. In: Proceedings of the 8th ACM international conference on Aspect-oriented software development. pp. 63–74.

White, A., 2002. SERP, an Open Source framework for manipulating Java bytecode. http://serp.sourceforge.net/.

Zaidman, A., 2006. Scalability solutions for program comprehension through dynamic analysis. In: Proceedings of the 10th European Conference on Software Maintenance and Reengineering. IEEE.